Lecture 09
Control Flow Instructions

Comp_Sci 213

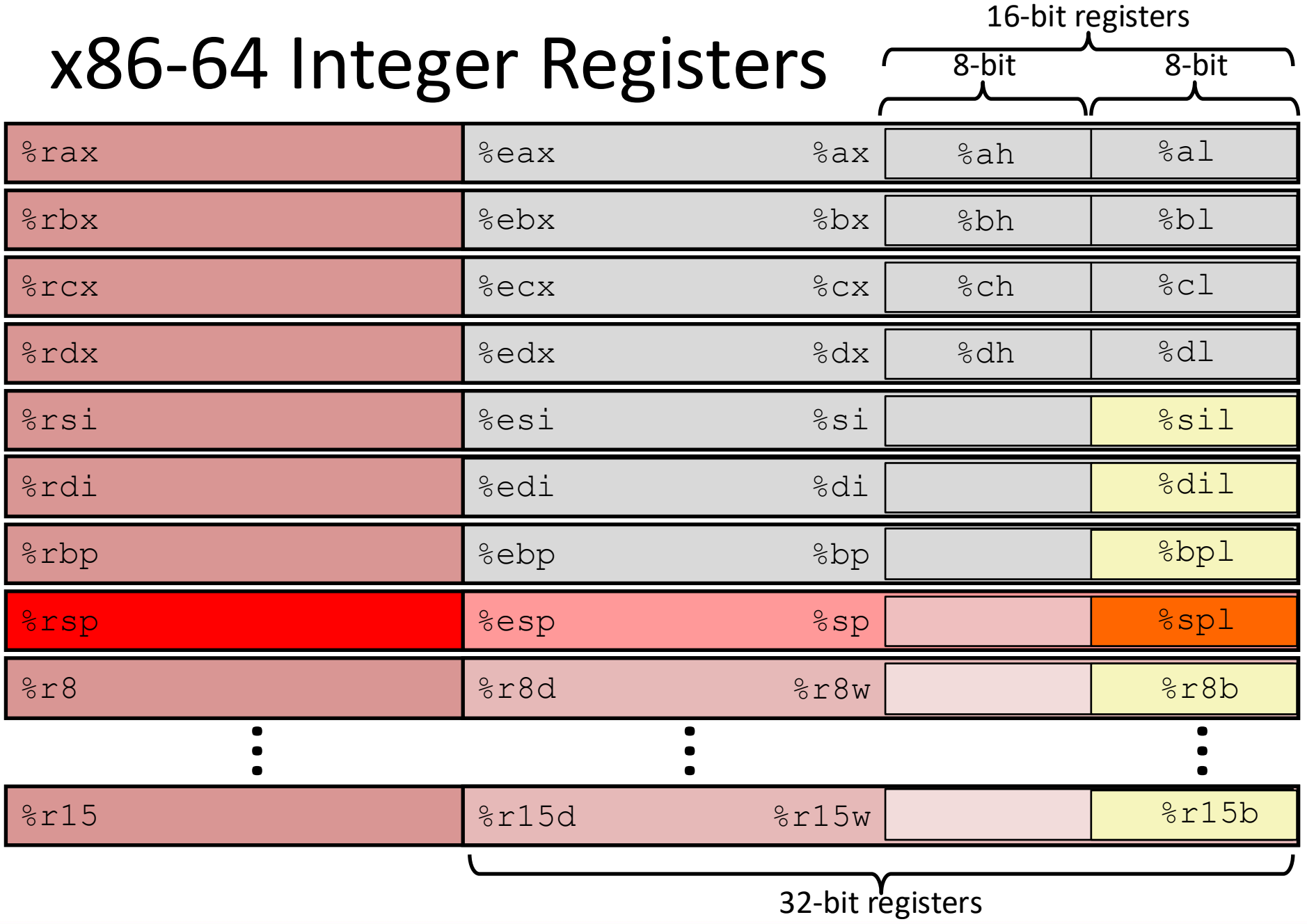
Introduction to Computer Systems

Monday, Oct. 21, 2026

Kate Smith – Fall 2026

Slides adapted from:
Ghena, St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

x86-64 Integer Registers



Topics

- Condition Codes (review)
- Branching (If/Else)
- Loops (Do While, While, For)
- Conditional Move

Topics

- **Condition Codes (review)**
- Branching (If/Else)
- Loops (Do While, While, For)
- Conditional Move

Condition codes

- Control is mediated via *Condition codes*
  - single-bit registers that record answers to questions about values
  - E.g., Is value x greater than value y? Are they equal? Is their sum even?
  - Let's keep "question" abstract for now. We'll see the details in a bit.
  - **Terminology**:
    - a bit is ***set*** if it is 1
    - a bit is ***cleared*** (or ***reset***) if it is 0

What are the condition codes?

- Condition codes on x86
  - **CF** Carry Flag (for unsigned)
  - **ZF** Zero Flag
  - **PF** Parity Flag
  - **SF** Sign Flag (for signed)
  - **OF** Overflow Flag (for signed)

  - Not an arbitrary set! By combining them, can keep track of answers to many useful questions! (We'll see exactly which in a bit.)

Explicitly Setting Condition Codes: Compare

- **`cmp{b,w,l,q}`** ***`Src2, Src1`***
- **`cmpq src,dst`** computes **`t = dst-src`**, ignoring the result
  - And sets condition codes along the way, like **`subq`** would!
  - Follows the rules we saw on the previous slide for arithmetic instructions
  - **Beware the order of the `cmp` operands!**

- Use cases
  - **ZF set** if **`dst == src`**
  - **SF set** if **`(dst-src) < 0`** (as signed), i.e., **`dst < src`** in a signed comparison!
  - **CF and OF** used mostly in combinations with others (see in a few slides)

Reading Condition Codes

- Cannot read condition codes directly; instead observe via instructions
  - And generally observe ***combinations*** of condition codes, not individual ones

- Example: the **`setX`** family of instructions
  - Write single-byte destination register based on combinations of condition codes
    - **`set{e, ne, s, …} D`** where D is a 1-byte register
    - Example: **`sete %al`**
      - means: **`%al`**=1 if flag ZF is set, **`%al`**=0 otherwise

Condition codes combinations

Suffix	Description	Condition
e	Equal / Zero	ZF
ne	Not Equal / Not Zero	~ZF
s	Negative	SF
ns	Nonnegative	~SF
g	Greater (Signed)	~(SF^OF)&~ZF
ge	Greater or Equal (Signed)	~(SF^OF)
l	Less (Signed)	(SF^OF)
le	Less or Equal (Signed)	(SF^OF)|ZF
a	Above (unsigned)	~CF&~ZF
b	Below (unsigned)	CF

Note: suffixes do not indicate operand sizes, but rather conditions

These same suffixes will come back when we see other instructions that read condition codes.

Expect to be run after a **`cmp`**

What do you need to know?

- 90%+ of the time
  - `cmp` instruction followed by `setX` instruction (or a branch)
  - **`cmpq src, dst`** computes **`dst-src`**, then throws away the result
  - Does NOT modify the destination register – be careful of operand order!
  - Think of as `dst X src`
    - `dst <= src` or `dst != src` etc.
- 10% or less of the time
  - Arbitrary arithmetic instruction sets the condition codes
    - Or `testq` sets the condition codes
  - Followed by a `setX` or branch
  - And you actually have to think about which condition codes are set to figure out what the assembly is doing, which can be challenging

`op src, dst`

Practice

- `set`**X** asks the question: "Is destination **X** source?"
  - Usually condition codes from "`cmp source, destination`"
  - Don't have to care about the exact values of the condition codes though
    - Just understand the logic

```
%sil = 0x01
%dil = 0xF0

cmp %sil, %dil
seta  %al
sete  %al
setge %al
```

SetX	Description
`sete`	Equal / Zero
`setne`	Not Equal / Not Zero
`sets`	Negative
`setns`	Nonnegative
`setg`	Greater (Signed)
`setge`	Greater or Equal (Signed)
`setl`	Less (Signed)
`setle`	Less or Equal (Signed)
`seta`	Above (unsigned)
`setb`	Below (unsigned)

`op src, dst`

Practice

- `set`**X** asks the question: "Is destination **X** source?"
  - Usually condition codes from "`cmp source, destination`"
  - Don't have to care about the exact values of the condition codes though
    - Just understand the logic

`%sil = 0x01`

`%dil = 0xF0`

```
cmp %sil, %dil
```

TRUE `seta  %al` # is `%dil` ABOVE `%sil` (unsigned)

FALSE `sete  %al` # is `%dil` EQUAL to `%sil`

FALSE `setge %al` # is `%dil` GREATER or EQUAL to `%sil` (signed)

Topics

- Condition Codes (review)
- **Branching (If/Else)**
- Loops (Do While, While, For)
- Conditional Move

What can instructions do?

- Move data: ✓
- Arithmetic: ✓
- **Transfer control**
  - Instead of executing next instruction, go somewhere else

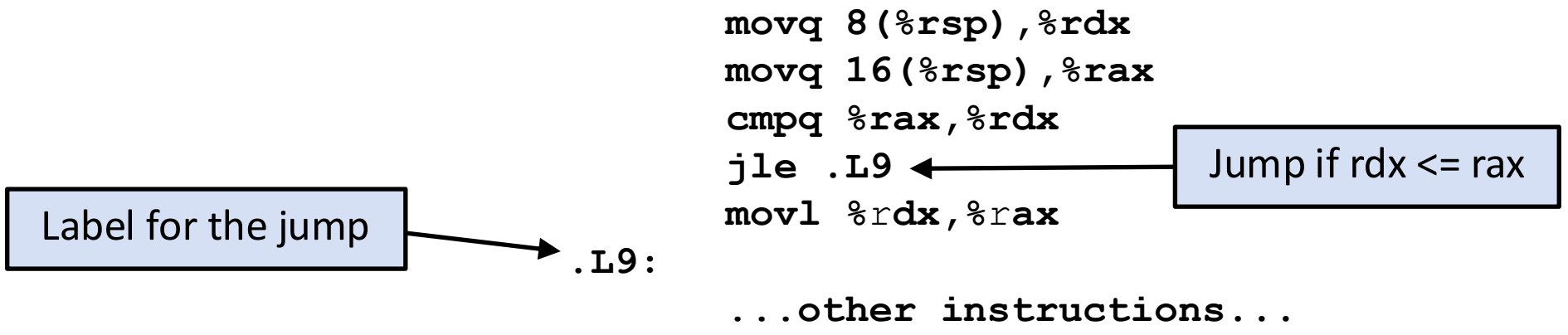
```
if (x > y)
    result = x-y;
else
    result = y-x;
```

```
while (x > y)
    result = x-y;
return result;
```

  - Sometimes we want to go from the red code to the green code
  - But the blue code is what's next!
  - Need to transfer control! Execute an instruction that is not the next one
  - And ***conditionally***, too! (i.e., based on a condition)

Breaking with sequential execution

- "Normal" execution follows instructions in listed (sequential) order
- To move to a different location – **jump**
  - Jump to different part of code depending on condition codes
  - Destination of a jump – a label: particular address at which we find code
  - Label addresses are determined when generating the object code



Jumping

- `jX` Instructions
  - Jump to different part of code depending on condition codes
  - **`jmp`** has two options
    - **Direct**: to a label (literal address)
    - **Indirect**: based on a register
    - Direct is by far the most common

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jae`	`~CF`	Above or Equal (unsigned)
`jb`	`CF`	Below (unsigned)
`...`	`...`	...

Documentation on all jump instructions: https://en.wikibooks.org/wiki/X86_Assembly/Control_Flow

Key idea: building C constructs with assembly

- Jump will let us build the flow control statements in C
  - If, While, For, Switch, etc.

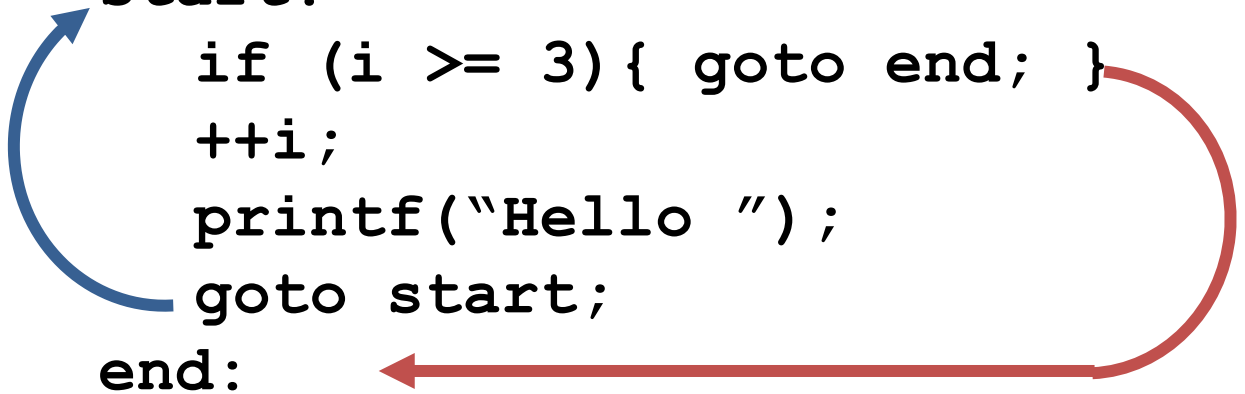
- But the translation isn't always obvious
  - Might switch ordering, or negate the logical condition
  - Maintains the same result when it runs, but easier for assembly

- Steps
  1. Transform C into something simpler (closer to assembly)
  2. Transform simpler C into assembly

A simpler approach is goto

- C allows **goto** as means of transferring control
  - Closer to machine-level programming style
  - Place labels wherever you want in code
  - Goto "jumps" to the referenced label
- Generally considered bad programming style
  - Makes it really difficult to understand what code is doing

```
    int i = 0;
start:
    if (i >= 3){ goto end; }
    ++i;
    printf("Hello ");
    goto start;
end:
    printf("World!\n");
```



Prints:

"Hello Hello Hello World!\n"

Conditional Branch Example

```
long absdiff(long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```



```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = (x <= y);
    if (ntest) { goto Else; }
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

- Translate an if statement into a “simpler” goto statement
  - Makes the if statement closer to machine code because goto can translate to jumps
  - Note: execution doesn’t stop at a label, it just keeps going

Conditional Branch Example

Goto Version

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = (x <= y);
    if (ntest) {goto Else;}
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

Asm Version

```
absdiff:
    cmpq    %rsi, %rdi   # cmp x:y
    jle     .L2          # x <= y
    movq    %rdi, %rax
    subq    %rsi, %rax
    jmp     .L3
.L2:                     # jle target
    movq    %rsi, %rax
    subq    %rdi, %rax
.L3:
    ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

Conditional Branch Example

Goto Version

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = (x <= y);
    if (ntest) {goto Else;}
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

Asm Version

```
absdiff:
    cmpq    %rsi, %rdi   # cmp x:y
    jle     .L2          # x <= y
    movq    %rdi, %rax
    subq    %rsi, %rax
    jmp     .L3
.L2:                     # jle target
    movq    %rsi, %rax
    subq    %rdi, %rax
.L3:
    ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

Conditional Branch Example

Goto Version

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = (x <= y);
    if (ntest) {goto Else;}
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

Asm Version

```
absdiff:
    cmpq    %rsi, %rdi   # cmp x:y
    jle     .L2          # x <= y
    movq    %rdi, %rax
    subq    %rsi, %rax
    jmp     .L3
.L2:                     # jle target
    movq    %rsi, %rax
    subq    %rdi, %rax
.L3:
    ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

Conditional Branch Example

Goto Version

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = (x <= y);
    if (ntest) {goto Else;}
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

Asm Version

```
absdiff:
    cmpq    %rsi, %rdi   # cmp x:y
    jle     .L2          # x <= y
    movq    %rdi, %rax
    subq    %rsi, %rax
    jmp     .L3
.L2:                     # jle target
    movq    %rsi, %rax
    subq    %rdi, %rax
.L3:
    ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

Conditional Branch Example

Goto Version

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = (x <= y);
    if (ntest) {goto Else;}
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

Asm Version

```
absdiff:
    cmpq    %rsi, %rdi   # cmp x:y
    jle     .L2          # x <= y
    movq    %rdi, %rax
    subq    %rsi, %rax
    jmp     .L3
.L2:                     # jle target
    movq    %rsi, %rax
    subq    %rdi, %rax
.L3:
    ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

Conditional Branch Example

Goto Version

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = (x <= y);
    if (ntest) {goto Else;}
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

Asm Version

```
absdiff:
    cmpq    %rsi, %rdi   # cmp x:y
    jle     .L2          # x <= y
    movq    %rdi, %rax
    subq    %rsi, %rax
    jmp     .L3
.L2:                     # jle target
    movq    %rsi, %rax
    subq    %rdi, %rax
.L3:
    ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

General “if-then-else” translation

C Code

```
if (test-expr)
   then-statement
else
   else-statement
```

Goto Version

```
   ntest = !(test-expr);
   if (ntest){
      goto Else;
   }
   then-statement;
   goto done;
Else:
   else-statement;
done:
```

- *test-expr* is an expression returning integer
  - = 0 interpreted as false, ≠0 interpreted as true
- Only one of the two statements is executed
  - i.e. only one of the two *branches* of code
- That’s one translation; there are others
  - E.g., flipping the order of the blocks instead of flipping the test
- Conditional expressions (`x ? y : z`) can use the same translation

If statement - bigger example

```
long test(long a, long
b){
   long c;
   if (a > b) {
      c = 1;
   } else if (a < b) {
      c = -1;
   } else {
      c = 0;
   }
   return c;
}
```

If statement - bigger example

a→%rdi, b→%rsi, c→%rax

```
long test(long a, long
b){
  long c;
  if (a > b) {
    c = 1;
  } else if (a < b) {
    c = -1;
  } else {
    c = 0;
  }
  return c;
}
```

```
  cmp %rsi, %rdi
  jle elif      # !(a > b)
  movq $1, %rax
  jmp end
elif:
  cmp %rsi, %rdi
  jge else      # !(a < b)
  movq $-1, %rax
  jmp end
else:
  movq $0, %rax
end:
  ret           # returns %rax
```

If statement - bigger example

a→%rdi, b→%rsi, c→%rax

```
long test(long a, long
b){
  long c;
  if (a > b) {
    c = 1;
  } else if (a < b) {
    c = -1;
  } else {
    c = 0;
  }
  return c;
}
```

```
  cmp %rsi, %rdi
  jle elif        # !(a > b)
  movq $1, %rax
  jmp end
elif:
  cmp %rsi, %rdi
  jge else        # !(a < b)
  movq $-1, %rax
  jmp end
else:
  movq $0, %rax
end:
  ret             # returns %rax
```

If statement - bigger example

a→%rdi, b→%rsi, c→%rax

```
long test(long a, long
b){
   long c;
   if (a > b) {
      c = 1;
   } else if (a < b) {
      c = -1;
   } else {
      c = 0;
   }
   return c;
}
```

```
   cmp %rsi, %rdi
   jle elif       # !(a > b)
   movq $1, %rax
   jmp end
elif:
   cmp %rsi, %rdi
   jge else       # !(a < b)
   movq $-1, %rax
   jmp end
else:
   movq $0, %rax
end:
   ret            # returns %rax
```

If statement - bigger example

a→%rdi, b→%rsi, c→%rax

```
long test(long a, long
b){
   long c;
   if (a > b) {
      c = 1;
   } else if (a < b) {
      c = -1;
   } else {
      c = 0;
   }
   return c;
}
```

```
   cmp %rsi, %rdi
   jle elif       # !(a > b)
   movq $1, %rax
   jmp end
elif:
   cmp %rsi, %rdi
   jge else       # !(a < b)
   movq $-1, %rax
   jmp end
else:
   movq $0, %rax
end:
   ret            # returns %rax
```

If statement - bigger example

a→%rdi, b→%rsi, c→%rax

```
long test(long a, long
b){
   long c;
   if (a > b) {
      c = 1;
   } else if (a < b) {
      c = -1;
   } else {
      c = 0;
   }
   return c;
}
```

```
   cmp %rsi, %rdi
   jle elif       # !(a > b)
   movq $1, %rax
   jmp end
elif:
   cmp %rsi, %rdi     # unnecessary
   jge else       # !(a < b)
   movq $-1, %rax
   jmp end
else:
   movq $0, %rax
end:
   ret            # returns %rax
```

Optimization (O1)

a→%rdi, b→%rsi, c→%rax

```
long test(long a, long
b){
   long c;
   if (a > b) {
      c = 1;
   } else if (a < b) {
      c = -1;
   } else {
      c = 0;
   }
   return c;
}
```

```
   movq $1, %rax
   cmp %rsi, %rdi
   jg end
   setl %al
   movzbq %al, %rax
   neg %rax
end:
   ret                  # returns %rax
```

What is the highlighted code block doing above?

Optimization (O1)

```
long test(long a, long
b){
   long c;
   if (a > b) {
      c = 1;
   } else if (a < b) {
      c = -1;
   } else {
      c = 0;
   }
   return c;
}
```

a→%rdi, b→%rsi, c→%rax

```
   movq $1, %rax
   cmp %rsi, %rdi
   jg end
   setl %al
   movzbq %al, %rax
   neg %rax
end:
   ret                  # returns %rax
```

else if and else together

What is the highlighted code block doing above?

Generates 0 (not less) or -1 (less)

Reminder: Moving data of different sizes

`op src, dst`

- "Vanilla" move can only move between source and destination of the same size
  - Larger → smaller: use the smaller version of registers, which truncates
  - Smaller → larger: extension! We have two options: zero-extend or sign-extend

Instruction	Effect	Description
movX S,D X ∈ {**q**, **l**, **w**, **b**}	D ← S	Copy quad-word (8B), long-word (4B), word (2B) or byte (1B)
movsXX S,D XX ∈ {**bw**, **bl**, **wl**, **bq**, **wq**, **lq**}	D ← SignExtend(S)	Copy sign-extended byte to word, byte to long-word, etc.
movzXX S,D XX ∈ {**bw**, **bl**, **wl**, **bq**, **wq**, **lq**}	D ← ZeroExtend(S)	Copy zero-extended byte to word, byte to long-word, etc.
cltq (convert long to quad)	**%rax** ← SignExtend(**%eax**)	Sign-extend %eax to %rax Identical to `movslq %eax, %rax`

Reminder: moving byte data

`op src, dst`

- Note the differences between `movb`, `movsbl` and `movzbl`
- Assume `%dl = 0xCD`, `%eax = 0x98765432`

```
movb %dl,%al
```

%eax = 0x987654CD

```
movsbl %dl,%eax
```

%eax = 0xFFFFFFCD

```
movzbl %dl,%eax
```

%eax = 0x000000CD

Indirect jump

- `jmp *0x40000(%rdi, %rdx, 8)`
  - Calculate memory address: `0x40000 + %rdi + 8*%rdx`
  - Then load value from that memory address
  - Treat value as an address and jump to it

- Indirect jumps jump to the address loaded from memory
  - Essentially a function pointer
  - Or used for a Jump Table: efficient switch statements
- The `*` lets you know that something tricky is going on
  - Displacement could be a label rather than a value

Topics

- Condition Codes (review)
- Branching (If/Else)
- **Loops (Do While, While, For)**
- Conditional Move

Loops

- C provides different looping constructs
  - **`while`, `do` … `while`, `for`**
- No corresponding instruction in machine code
- Most compilers
  1. Transform general loops into **`do` … `while`**

```
do
   body-statement
   while (test-expr);
```

Do-while:
Same idea as a while loop, but the body always runs at least once

  2. Rewrite that with goto
  3. Then compile result into machine code

“Do-While” Loop Example

- Running example: count number of 1s in x (“popcount”)
  - We’ll write it with different kinds of loops
  - What the body of the loop does is not our focus; we’ll just ignore it

C Code

```
long pcount_do
   (unsigned long x)
{
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

“Do-While” Loop Compilation

- Step 1: translate do-while into a gotos
- Use conditional branch to either continue looping or to exit loop

C Code

```
long pcount_do
   (unsigned long x)
{
   long result = 0;
   do {
      result += x & 0x1;
      x >>= 1;
   } while (x);
   return result;
}
```

Goto Version

```
long pcount_goto
   (unsigned long x)
{
   long result = 0;
loop:
   result += x & 0x1;
   x >>= 1;
   if (x) {goto loop;}
   return result;
}
```

“Do-While” assembly translation

Register	Use(s)
%rdi	Argument x
%rax	result

Goto Version

```
long pcount_goto
   (unsigned long x) {
   long result = 0;
 loop:
   result += x & 0x1;
   x >>= 1;
   if (x) {goto loop;}
   return result;
}
```

```
      movq    $0,%rax      #  result = 0
.L2:                       # loop:
      movq    %rdi,%rdx
      andq    $1,%rdx      #  t = x & 0x1
      addq    %rdx,%rax    #  result += t
      shrq    %rdi         #  x >>= 1
      jne     .L2          #  if (x) goto loop
      rep; ret
```

“Do-While” assembly translation

Which instruction sets the condition codes for `jne`?

Logical shift right (`shrq`)

– the comparison is whether the result is 0 or not

Register	Use(s)
`%rdi`	Argument x
`%rax`	result

Goto Version

```
long pcount_goto
   (unsigned long x) {
   long result = 0;
 loop:
   result += x & 0x1;
   x >>= 1;
   if (x) {goto loop;}
   return result;
}
```

```
      movq    $0,%rax      #  result = 0
.L2:                       # loop:
      movq    %rdi,%rdx
      andq    $1,%rdx      #  t = x & 0x1
      addq    %rdx,%rax    #  result += t
      shrq    %rdi         #  x >>= 1
      jne     .L2          #  if (x) goto loop
      rep; ret
```

“Do-While” assembly translation

Register	Use(s)
%rdi	Argument x
%rax	result

Goto Version

```
long pcount_goto
   (unsigned long x) {
   long result = 0;
 loop:
   result += x & 0x1;
   x >>= 1;
   if (x) {goto loop;}
   return result;
}
```

```
      movq    $0,%rax      #  result = 0
.L2:                       # loop:
      movq    %rdi,%rdx
      andq    $1,%rdx      #  t = x & 0x1
      addq    %rdx,%rax    #  result += t
      shrq    %rdi         #  x >>= 1
      jne     .L2          #  if (x) goto loop
      rep; ret
```

rep instruction repeats string operations following it…**why is this?!**

“Do-While” assembly translation

`rep; ret` uses `rep` as a no-op (a.k.a nop, an operation that does nothing)

- Example of a compiler optimization that you might run into in real assembly code
- AMD recommends this to speed up execution when there is a jump before a return
- See CE361 and CE452 for more details (Computer Architecture courses)

Register	Use(s)
`%rdi`	Argument x
`%rax`	result

Goto Version

```
long pcount_goto
   (unsigned long x) {
   long result = 0;
 loop:
   result += x & 0x1;
   x >>= 1;
   if (x) {goto loop;}
   return result;
}
```

```
      movq    $0,%rax      #  result = 0
.L2:                       # loop:
      movq    %rdi,%rdx
      andq    $1,%rdx      #  t = x & 0x1
      addq    %rdx,%rax    #  result += t
      shrq    %rdi         #  x >>= 1
      jne     .L2          #  if (x) goto loop
      rep; ret
```

`rep` instruction repeats string operations following it…**why is this?!**

General “Do-While” Translation

- Body: {
  Statement$_1$;
  Statement$_2$;
  …
  Statement$_n$;
  }

C Code

```
do
  Body
  while (Test);
```

Goto Version

```
loop:
  Body
  if (Test){
    goto loop
  }
```

- Test returns integer
  - = 0 interpreted as false
  - ≠ 0 interpreted as true

General “While” Translation #1

- “Jump-to-middle” translation
- Most straightforward match to how “while” works

While version

```
while (Test){
   Body
}
```



Goto Version

```
   goto test;
loop:
   Body
test:
   if (Test ){
      goto loop;
   }
```

While Loop Example #1

C Code

```
long pcount_while
   (unsigned long x)
{
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Jump to Middle

```
long pcount_goto_jtm
   (unsigned long x)
{
  long result = 0;
  goto test;
 loop:
  result += x & 0x1;
  x >>= 1;
 test:
  if(x) {goto loop;}
  return result;
}
```

- Initial goto starts loop at test

Comparing while to do-while

While with goto (jump to middle)

```
long pcount_while_goto_jtm
   (unsigned long x)
{
  long result = 0;
  goto test;
 loop:
  result += x & 0x1;
  x >>= 1;
 test:
  if(x) {goto loop;}
  return result;
}
```

Do While with goto

```
long pcount_dowhile_goto
   (unsigned long x)
{
  long result = 0;

 loop:
  result += x & 0x1;
  x >>= 1;

  if (x) {goto loop;}
  return result;
}
```

General “While” Translation #2

While version

```
while (Test)
  Body
```



Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

- “Do-while” conversion
- More optimized compiler translation
  - “Guarded” loop

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

“While” Loop Example #2

C Code

```
long pcount_while
   (unsigned long x)
{
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Goto Version

```
long pcount_goto_dw
   (unsigned long x)
{
  long result = 0;
  if (!x) {goto done;}
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) {goto loop;}
 done:
  return result;
}
```

- Initial conditional guards entrance to loop

Comparing jump-to-middle and guarded-do-while

While with goto (jump to middle)

```
long pcount_while_goto_jtm
   (unsigned long x)
{
  long result = 0;
  goto test;
 loop:
  result += x & 0x1;
  x >>= 1;
 test:
  if(x) {goto loop;}

  return result;
}
```

While with goto (guarded do-while)

```
long pcount_goto_dw
   (unsigned long x)
{
  long result = 0;
  if (!x) {goto done;}
 loop:
  result += x & 0x1;
  x >>= 1;

  if(x) {goto loop;}
 done:
  return result;
}
```

"For" Loop Form

General Form

```
for (Init; Test; Update)
    Body
```

```
#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit = (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit = (x >> i) & 0x1;
  result += bit;
}
```

“For” → “While” → “Do-While” → “Goto”

For Version

```
for (Init; Test; Update)
    Body
```

While Version

```
Init;
while (Test) {
    Body
    Update ;
}
```

Do-While Version

```
  Init;
  if (!Test)
    goto done;
  do {
    Body
    Update ;
  } while (Test)
done:
```

Goto Version

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update ;
  if (Test)
    goto loop;
done:
```

"For" Loop Conversion Example

C Code

```
#define WSIZE 8*sizeof(int)
long pcount_for(unsigned x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned bit =
       (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Goto Version

```
#define WSIZE 8*sizeof(int)
long pcount_for_gt(unsigned x)
{
  size_t i;
  long result = 0;
  i = 0;                    Init
  if (!(i < WSIZE))         !Test
    goto done;
 loop:
  {
    unsigned bit =          Body
       (x >> i) & 0x1;
    result += bit;
  }
  i++;                      Update
  if (i < WSIZE)            Test
    goto loop;
 done:
  return result;
}
```

Assembly to loop

What does this function do?

```
my_function:     # %rdi is argument1
   mov $0, %rax
   mov $0, %rbx
   test %rdi, %rdi
   je end
loop:
   add %rbx, %rax
   add $1, %rbx
   cmp %rdi, %rbx
   jne loop
end:
   ret           # returns %rax
```

Note: this seems strange, but remember, `test` performs bitwise AND and sets flags based on the result. ZF will be set if `%rdi` is zero. SF will also be set if `%rdi` is negative!

Assembly to loop

```
my_function:    # %rdi is argument1
  mov $0, %rax  # clear variables
  mov $0, %rbx
  test %rdi, %rdi
  je end
loop:
  add %rbx, %rax
  add $1, %rbx
  cmp %rdi, %rbx
  jne loop
end:
  ret           # returns %rax
```

Assembly to loop

```
my_function:     # %rdi is argument1
  mov $0, %rax   # clear variables
  mov $0, %rbx
  test %rdi, %rdi
  je end         # skip loop if %rdi is 0
loop:
  add %rbx, %rax
  add $1, %rbx
  cmp %rdi, %rbx
  jne loop
end:
  ret            # returns %rax
```

Assembly to loop

```
my_function:     # %rdi is argument1
  mov $0, %rax   # clear variables
  mov $0, %rbx
  test %rdi, %rdi
  je end         # skip loop if %rdi is 0
loop:
  add %rbx, %rax     # %rax += %rbx
  add $1, %rbx   # %rbx += 1
  cmp %rdi, %rbx
  jne loop
end:
  ret            # returns %rax
```

Assembly to loop

```
my_function:     # %rdi is argument1
  mov $0, %rax   # clear variables
  mov $0, %rbx
  test %rdi, %rdi
  je end         # skip loop if %rdi is 0
loop:
  add %rbx, %rax     # %rax += %rbx
  add $1, %rbx   # %rbx += 1
  cmp %rdi, %rbx
  jne loop       # while %rbx != %rdi
end:
  ret            # returns %rax
```

Assembly to loop

```
long my_function(long rdi) {
  long rax = 0;
  long rbx = 0;

  while (rbx != rdi) {
    rax += rbx;
    rbx += 1;
  }

  return rax;
}
```

```
my_function:    # %rdi is argument1
  mov $0, %rax  # clear variables
  mov $0, %rbx
  test %rdi, %rdi
  je end        # skip loop if %rdi is 0
loop:
  add %rbx, %rax    # %rax += %rbx
  add $1, %rbx  # %rbx += 1
  cmp %rdi, %rbx
  jne loop      # while %rbx != %rdi
end:
  ret           # returns %rax
```

Assembly to loop

```
long my_function(long rdi) {
  long rax = 0;
  long rbx = 0;

  while (rbx != rdi) {
    rax += rbx;
    rbx += 1;
  }

  return rax;
}
```

```
long my_function(long max) {
    long result = 0;
    for (long i=0; i<max; i++){
        result += i;
    }

    return result;
}
```

```
my_function:    # %rdi is argument1
  mov $0, %rax  # clear variables
  mov $0, %rbx
  test %rdi, %rdi
  je end        # skip loop if %rdi is 0
loop:
  add %rbx, %rax    # %rax += %rbx
  add $1, %rbx  # %rbx += 1
  cmp %rdi, %rbx
  jne loop      # while %rbx != %rdi
end:
  ret           # returns %rax
```

Note: ret doesn't "know" anything about `%rax`…when you return, all values in `%rax`, `%rbx`, and `%rdi` remain in registers for future processing. That's when `%rax` might be used!

Topics

- Condition Codes (review)
- Branching (If/Else)
- Loops (Do While, While, For)
- **Conditional Move**

The Problem with Conditional Jumps

- Conditional jumps = conditional *transfer of control*
  - i.e., forget what you thought you were going to do, do this other thing instead
- Modern processors like to do work "ahead of time"
  - Keywords: ***pipelining, branch prediction, speculative execution***
  - Transfer of control may mean throwing that work away
    - That's inefficient
- Solution: conditional *moves*
  - We still get to do something conditionally
  - But no transfer of control necessary
  - "Ahead of time" work can always be kept

Conditional Moves

cmovX	Description
`cmove S, D`	equal / Zero
`cmovne S, D`	not equal / Not zero
`cmovs S, D`	negative
`cmovns S, D`	nonnegative
`cmovg S, D`	greater (Signed)
`cmovge S, D`	greater or equal (Signed)
`cmovl S, D`	less (Signed)
`cmovle S, D`	less or equal (Signed)
`cmova S, D`	above (Unsigned)
`cmovae S, D`	above or equal (Unsigned)
`cmovb S, D`	below (Unsigned)
`cmovbe S, D`	below or equal (Unsigned)

D ← S only if test condition is true

Conditional Move Example

```
long absdiff(long x, long y)
{
  long res;
  if (x > y)
    res = x-y;
  else
    res = y-x;
  return res;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

```
absdiff:
    movq    %rdi, %rax   # res = x
    subq    %rsi, %rax   # res = x-y
    movq    %rsi, %rdx
    subq    %rdi, %rdx   # alt = y-x
    cmpq    %rsi, %rdi   # cmp x:y
    cmovle  %rdx, %rax   # if x<=y, res = alt
    ret
```

No branching!

Must compute both results, though, which is not always possible or desirable…

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

- A `cmov` requires that both values get computed
- Could trigger a fault (compiler must use jumps instead)

Computations with side effects

```
val = x > 0 ? x++ : x--;
```

- Both values get computed
- Needs use extra temporary registers to hold intermediate results

If, else if, else – optimized (O3)

```
long test(long a, long
b){
   long c;
   if (a > b) {
      c = 1;
   } else if (a < b) {
      c = -1;
   } else {
      c = 0;
   }
   return c;
}
```

a→%rdi, b→%rsi, c→%rax

```
movq $0, %rax # clear reg
cmp %rsi, %rdi
movq $1, %rdx
setl %al
neg %rax
cmp %rsi, %rdi
cmovg %rdx, %rax
ret                  # returns %rax
```

else if and else together (%al is %rax)

select output

Midterm review

Midterm exam

- Midterm on Monday, Oct. 26
  - If you need accommodations, please contact AccessibleNU by today!!
- In class, pencil and paper
- You can have 1 cheat sheet, double sided
  - Handwritten or typed!
- You can have a calculator (no phones/calculators that connect to internet)
- Our exam will be different from the other 213 section

More `leaq` examples

`leaq` computes the effective address `Sb + Si*S + D`

Adding two registers:

- `leaq (%rdi,%rsi), %rax # rax = x + y`
- `leaq (%rdx,%rcx), %r8 # r8 = z + w`

Adding with a scaled index

- `leaq (%rdi,%rsi,2), %rdx  # rdx = x + 2*y`
- `leaq (%rdi,%rsi,4), %rax  # rax = x + 4*y`
- `leaq (%rdi,%rsi,8), %rax  # rax = x + 8*y`

More `leaq` examples

`leaq` computes the effective address `Sb + Si*S + D`

Multiplying by constants (no base register):

- `leaq (,%rdi,8), %rax # rax = 8*x`

Multiplying by constants (base = index)

- `leaq (%rdi,%rdi,4), %rax # rax = 5*x ; (x + 4x)`
- `leaq (%rdi,%rdi,8), %rax # rax = 9*x ;(x + 8x)`
- `leaq (%rsi,%rsi,2), %rsi # rsi = 3*y ; (y + 2y)`

More `leaq` examples

`leaq` computes the effective address `Sb + Si*S + D`

Adding a constant offset:

- `leaq 7(%rdi,%rdi,4), %rax # rax = 5*x + 7`
- `leaq -12(%rdx), %rcx # rcx = z - 12`

Composing (helpful for multiplications that can't be produced by a single scale):

```
leaq (%rdi,%rdi,4), %rax # rax = 5*x
leaq (%rax,%rax), %rax # rax = 2*(5*x) = 10*x
```

Other helpful topics to review

Review class material from lecture 1 to today. Topics to emphasize:

- Rules for floating point
- Big-picture concepts from labs
- Signed / unsigned numbers, binary & hex
- Binary arithmetic
- Boolean algebra
- Bitwise operators
- Endianness
- Overflow
- Register concepts
- Understanding assembly
- Clock frequency / period